Thesis to obtain the degree of a Masters of Science by Research.

Applications of Bee Colony Optimization

Aidan Chalk

September 2012

Supervisors

Dr. Matthew Johnson

Prof. Iain Stewart

Contents

Abstract

Many computationally difficult problems are attacked using non-exact algorithms, such as approximation algorithms and heuristics. This thesis investigates an example of the latter, Bee Colony Optimization, on both an established optimization problem in the form of the Quadratic Assignment Problem and the FireFighting problem, which has not been studied before as an optimization problem. Bee Colony Optimization is a swarm intelligence algorithm, a paradigm that has increased in popularity in recent years, and many of these algorithms are based on natural processes.

We tested the Bee Colony Optimization algorithm on the QAPLIB library of Quadratic Assignment Problem instances, which have either optimal or best known solutions readily available, and enabled us to compare the quality of solutions found by the algorithm. In addition, we implemented a couple of other well known algorithms for the Quadratic Assignment Problem and consequently we could analyse the runtime of our algorithm.

We introduce the Bee Colony Optimization algorithm for the FireFighting problem. We also implement some greedy algorithms and an Ant Colony Optimization algorithm for the FireFighting problem, and compare the results obtained on some randomly generated instances.

We conclude that Bee Colony Optimization finds good solutions for the Quadratic Assignment Problem, however further investigation on speedup methods is needed to improve its performance to that of other algorithms. In addition, Bee Colony Optimization is effective on small instances of the FireFighting problem, however as instance size increases the results worsen in comparison to the greedy algorithms, and more work is needed to improve the decisions made on these instances.

Acknowledgments

First, I would like to thank Matthew Johnson who has been my supervisor during my Masters project. His direction and comments have been of great help in my first forays into academic research.

I would also like to thank Iain Stewart, who first got me interested in the field and supervised my undergraduate dissertation in a similar area.

Thirdly, I would like to thank Alan Heppenstall for deciphering PageRank and helping my own brief exploration into the area. Also Helen Benjamin for repeatedly explaining to us simple linear algebra.

Finally, I would like to thank the Department of Engineering and Computer Sciences for granting me the scholarship and opportunity to undertake this research.

Declaration of Copyright

1. Introduction

This thesis explores the application of Bee Colony Optimization (a swarm intelligence metaheuristic) to a pair of combinatorial optimization problems, the Firefighting problem and the Quadratic Assignment problem. These problems are formally defined in chapter 2 which also outlines the Bee Colony Optimization algorithm and a variety of graph-theoretic measures we use later in the thesis.

Chapter 3 discusses the application of Bee Colony Optimization to the FireFighting problem and analyses some results on some randomly generated graphs, compared to results obtained by a pair of greedy algorithms and an implementation of Ant Colony Optimization.

Chapter 4 outlines Bee Colony Optimization for the Quadratic Assignment Problem, testing the algorithm on most instances in QAPLIB (a standard library of Quadratic Assignment Problem instances, [QAP]) and attempting tweaks on the algorithm to improve the runtime.

The remainder of this chapter discusses the central concepts of Combinatorial optimization, Complexity theory and Graph theory (and Graph classes).

Combinatorial Optimization

Combinatorial optimization studies problems in which we aim to select the best possible solution from a search space. Usually, we study problems in which exhaustive search is not a viable method for obtaining this solution, and often these problems have an equivalent **NP**-hard decision problem (**NP**-hard means at least as difficult as **NP**-complete and so if $\mathbf{P} \neq \mathbf{NP}$ there is no polynomial-time algorithm for **NP**-hard decision problems, and so no polynomial-time algorithm for the equivalent optimization problem). Instead of using exact algorithms for these problems (which we cannot compute in polynomial-time), we often use heuristics or local search algorithms, which we expect to give near-optimal solutions in polynomial running time.

These optimization problems arise in a large number of disciplines, such as routing, AI and engineering so creating effective algorithms that can be applied to many problems is ideal. If these problem-independent methodologies rely on iterative improvement they are known as *metaheuristics*.

Many *metaheuristics* have been adapted from natural processes, such as Genetic Algorithms (GAs) and Ant Colony Optimization (ACO), and are amongst the best algorithms for a variety of problems ([DS04]).

Complexity Theory

Complexity theory involves categorizing decision problems according to their computational difficulty. One of the most important decision problem categorizations is determining whether a problem is in **P** or if the problem is **NP**-complete. Problems in **P** can be solved in polynomial time (ie. there is an algorithm that can solve a

problem of input size n in time $O(n^c)$, where c is a constant), and are generally accepted to be computationally tractable. **NP**-complete problems cannot be solved in polynomial time unless $\mathbf{P} = \mathbf{NP}$, and are known as computationally hard problems. It is often useful to restrict the input for **NP**-complete problems to obtain polynomial-time algorithms for inputs with specific properties. A simple example of such a restriction is for the **NP**-complete problem, SATISFIABILITY. If we restrict the input so that each clause is of maximum size 2, we have the problem 2-SATISFIABILITY, which is known to be in **P**. More importantly for this thesis, this can also be applied to graph problems by specifying our input is of a specific class of graph, and we can often use the properties of the graph class to help solve the problem in polynomial time.

Graph Theory

Graph theory is the study of graphs, each defined by a vertex set, V, and an edge set, E. For an directed graph (or digraph) an edge is defined to be an ordered pair, (a, b), where a and b are elements of the vertex set, each pair defining an edge from a to b. To define the class of undirected graph, each edge is simply a unordered pair, (a, b), each representing an edge from both a to b and b to a. Within these classes, there are a large number of other classes, which have specific properties. For example, a *bipartite* graph has the property that its vertex set, V, can be split into 2 sets V_1, V_2 such that each edge (a, b) in the edge set E has the property that $a \in V_1$ iff $b \in V_2$ or $a \in V_2$ iff $b \in V_1$.

2. Literature Review

2.1. Bee Colony Optimization

Bee Colony Optimization is a *metaheuristic* based on the natural foraging behaviours of bees. When a bee successfully finds food, it returns to the hive and communicates the position and distance to the food source to hivemates via a waggle dance, which was not understood until decoded by Karl von Frisch in 1974 ([vF74]). The waggle dance is a figure of eight dance, and communicates direction as the dance's angle to the sun and the length of the central section of the dance is directly proportional to the distance to the food. Other hivemates then have a choice, they can choose to follow another bees' dance and fly to the same food source, where they may find more food sources, or to explore randomly (which in nature is very rare). Once the dancer has completed its dance it can either observe another bees dance, or return to the food source it advertised.

Bee System was first introduced in [LT01], which led to the development of Bee Colony Optimization and its application to the Traveling Salesman Problem (TSP) in [WLC08a]. The *metaheuristic* is a swarm intelligence approach, meaning it is characterized by individuals doing repetitive actions and a simple communication method between individuals, resulting in iterative improvement of solution quality. Bee Colony Optimization has been used to attack a variety of problems, including

TSP [WLC08b, WLC09a, WLC09b] and the p-median problem [TDS11].

The Bee Colony Optimization *metaheuristic* has 4 steps per iteration, and the algorithm iterates until some condition is met. The algorithm has a set of virtual bees, the number dependent on the problem and problem instance, though between 20 and 50 is normal. The input is first read into memory, usually as some number of matrices and the algorithm starts. Each iteration contains the following four steps:

1. **Solution Construction**. Each bee constructs a solution using the solution chosen in step 4 of the previous iteration (unless it is the first iteration, which is discussed later) and some problem-specific property (such as edge length in TSP).
2. **Daemon Actions**. Problem-specific actions and local searches are run in this phase to improve the solutions. The combination of constructive and local search methods helps strengthen results.
3. **Advertise *'Waggle Dance'***. Bees advertise good quality solutions to each other for use in later iterations.
4. **Follow *'Waggle Dance'***. Each bee constructs solutions based upon a previously constructed solution. In this step, each acquires such a solution, either by using the solution it constructed in the previous iteration, or by following a good solution created by another bee.

The stopping condition is commonly either a specified number of iterations, or a number of iterations without improvement.

Solution Construction

The solution construction section of the algorithm is the main area that needs investigating when applying the *metaheuristic* to a new problem. The construction of

the solution S, uses information from a previous iteration in the form of a "followed solution" S^*, where this solution is selected from the bee's previous solution, or copied from a '*Waggle Dance*' of another bee in step 4 of the previous iteration.

The solutions are constructed one element at a time (so for TSP the solution is constructed by selecting one node at a time). $P_{i,j}$ is the probability that element j is added to the solution as the n^{th} element where element i was the $(n-1)^{th}$ element. We also let θ denote the element that followed i in S^*, λ is a predefined constant in the range $[0, 1]$, and $A_{i,n}$ is the set of available choices from i (the choice made at time $n-1$ during this solution construction phase by this bee, eg. in TSP every vertex not yet chosen) at time n. At time 0 we say the bee is at the hive, and so the set $A_{i,n}$ is the set of all possible choices. We then define $P_{i,j}$ as:

$$P_{i,j} = \frac{p_{i,j}^{\alpha}\eta_{i,j}^{\beta}}{\sum_{j \in A_{i,n}} p_{i,j}^{\alpha}\eta_{i,j}^{\beta}} \tag{2.1}$$

where η is some problem specific value (such as inverse edge length in TSP, so j is more likely to be chosen after i if there is a short edge between them) and

$$p_{i,j} = \begin{cases} 0, & j \notin A_{i,n}, \\ \lambda, & j = \theta, \theta \in A_{i,n}, \\ \frac{1-\lambda}{|A_{i,n}|-1}, & j \neq \theta, \theta \in A_{i,n} \\ \frac{1}{|A_{i,n}|}, & \theta \notin A_{i,n}. \end{cases},$$

$p_{i,j}$ gives a λ chance of choosing θ if that choice is still valid (deviation from the followed solution in previous selections might prevent this being a valid choice), and a $1-\lambda$ chance of deviating, spread evenly across other allowed selections. If the selection made by the followed solution is not valid, the probability is evenly spread across all allowed selections. $P_{i,j}$ is then calculated by modifying these probabilities

by some problem specific measure $\eta_{i,j}$, which attempts to quantify how good each selection might be, and helps the algorithm move away from following poor solutions.

The first iteration of the algorithm is another special case as we have no followed solution, so θ doesn't exist. Other algorithms (such as Ant Colony Optimization) use special initialisation rules, however Bee Colony Optimization defines $p_{i,j} = \frac{1}{n}$ during this iteration, as $\theta \notin A_{i,n}$.

We use the values of α, β to encourage bees to either follow previous iteration results more closely (by increasing the value of $\frac{\alpha}{\beta}$) or to pay more attention to the problem specific values (by decreasing the value of $\frac{\alpha}{\beta}$).

We illustrate the definitions by considering a simple example from TSP. In TSP we have our distance matrix D where $d_{i,j}$ is the edge weight of edge (i, j), and our solution is an ordering of nodes to make up a Hamiltonian cycle. [WLC08a] uses $\eta_{i,j} = \frac{1}{d_{i,j}}$ as the measure of edge quality. During the first iteration the algorithm creates solution entirely dependent on the edge weights, with lower edge weights more likely to be selected. The move from the hive at time 0 is picked entirely at random as $\eta_{i,j}$ is undefined from the hive in the first iteration, and in later iterations is picked according by calculating $P_{i,j}$ as though $\eta_{i,j} = 1$ for all vertices (so favouring the θ selection). For TSP, λ lies in the range $[0.95, 0.99]$ as experiments found it allowed enough variation through solution construction to work effectively with the non-constructive sections of the algorithm.

If we take the graph with distance matrix

$$D = \begin{matrix} 0 & 1 & 2 & 4 \\ 1 & 0 & 5 & 8 \\ 2 & 5 & 0 & 12 \\ 4 & 8 & 12 & 0 \end{matrix}$$

$\lambda = 0.95$, $\alpha = 1$, $\beta = 2$ and assume we have a bee with followed solution $[1, 3, 4, 2]$. We initially move from the hive with probabilities $[0.95, 0.017, 0.017, 0.017]$. Assuming we move from the hive to node 1, $p_{i,j}$ is then $[0, 0.025, 0.95, 0.025]$, and $\eta_{i,j}$ is $[0, 1, 0.5, 0.25]$. $P_{i,j}$ is then $[0, 0.095, 0.90, 0.005]$, and let's assume we follow the high probability and add node 3 to our solution, which is now $[1, 3,]$. The algorithm repeats this again, getting $P_{i,j}$ values of $[0, 0.23, 0, 0.76]$. Clearly selecting node 2 next would give us a better solution, and our equation has pushed us towards selecting this node next, though it still favours node 4 substantially. However if we increase β to 5, this $P_{i,j}$ becomes $[0, 0.81, 0, 0.19]$, showing the importance of settings these parameters correctly for an instance.

Daemon Actions

During Daemon Actions, we attempt to improve the constructed solutions using quick local search algorithms before advertising them to the rest of the colony. For many problems this can help avoid spending too much time constructing solutions far from the optimal. For example, in TSP, use of 2-opt (a simple algorithm that breaks 2 edges in the solution, (a, b) and (c, d), and replaces them with (a, d) and (c, b) if this results in an improvement in quality of the solution) can quickly reduce the search space to the area near the optimal solution, resulting in better runtime [WLC08b].

Waggle Dance Phases

Bees only advertise solutions if they found their personal best solution.

Bees that find solutions significantly better than the average found in the previous iteration are highly likely (usually significantly greater than 80% [WLC08a]) to use

their own solution for a guide instead of choosing another bee's solution to follow. If the quality of the solution found by the bee is worse, the chance of following another bees' solution increases, as it seems less likely that iterating on weaker solutions is going to improve the overall solution.

Another important parameter is the number of iterations a good solution is advertised for. [WLC08b] use 3 formulae to define the duration, D_i, for the TSP:

$$D_i = K \cdot \frac{Pf_i}{Pf_{colony}}$$

where

$$Pf_i = \frac{1}{L_i},\ L_i = \text{tour length}$$

$$Pf_{colony} = \frac{1}{N_{Bee}} \sum_{i=1}^{N_{Bee}} Pf_i$$

K is a user defined constant and N_{Bee} is the number of bees in the colony.

Finally, if no bees have danced for a series of iterations then the value of the best solution found by each bee is weakened by a user-defined percentage, known as a memory adjustment policy. This prevents stagnation of the search as the bees begin to find new solutions believed to be their personal best and so waggle dancing begins again, which is important as it allows the algorithm to investigate more of the search space. Due to the local search procedure, this should not occur until

the solutions found are near-optimal, enabling increased search especially around potential optimals.

Fragmentation State Transition Rule

[WLC09a] introduces the Fragmentation State Transition Rule (FSTR) for Bee Colony Optimization on the TSP. They note that good solutions for a specific TSP instance are likely to have sections that are similar, eg. One solution (solution A) may be $[1, 2, 4, 5, 6, 7, 3]$, and another (solution B) $[3, 7, 6, 2, 4, 5, 1]$. Solutions that share most edge choices must have similar quality as the only differences in the example solutions are the edges $1, 2$ and $5, 6$ are replaced with the edges $1, 5$ and $2, 6$ (assuming the problem is symmetric - the FSTR has not been tested as far as we know for asymmetric TSP). They realised that due to the way swarm intelligence algorithms work, utilising both local search and constructive solutions, they only need to change a few edges (if any) to potentially get improvement from the local search. They then changed the transition rule as follows: before beginning the constructive phase of the algorithm, split our followed solution into fragments of a few cities, eg for solution A above $[1], [2, 4, 5], [6, 7, 3]$ could be our fragments. The fragments can either be of a fixed size or of a random size - this is implementation dependent. Now start constructing the solution as before - except only travel to cities at either end of a fragment - so in this example from the hive the choices are $[1, 2, 5, 6, 3]$. If node 1 is selected, the solution at this stage would be simply $[1]$, if node 2 is selected then it would be $[2, 4, 5]$ or if node 3 is selected, $[3, 7, 6]$. This reduces the number of selections needed in this example to 3 from 6, and the maximum number of probability calculations to 11 from 27. This should reduce the runtime of the constructive phase of the algorithm significantly, and as long as the size of fragments is correct the results should not be affected. The size of fragments

tends to be chosen experimentally.

Frequency Based Pruning Strategy

Another improvement for Bee Colony Optimization introduced in [WLC09b] is the Frequency Based Pruning Strategy (FBPS). While the FSTR presented above seeks to reduce the runtime of the *constructive segment* of the algorithm, the FBPS seeks to reduce the runtime of the *local search* section. However, unlike the FSTR which aims to reduce the runtime of every solution construction, the FBPS seeks to reduce runtime by running fewer local searches. The FBPS is based on building blocks, which for TSP could be edges used in solutions. Let's assume we have a 7-city TSP problem, we then need a 7x7 matrix, $\mathcal{H}$, initially filled with zeros.

We start running the algorithm without alterations, and our first found solution is $[A, B, C, D, E, F, G]$. We then need to update $\mathcal{H}$ with the building blocks used in the solution, so for this solution we need to update $AB, BC, ...$ and the reverse $BA, CB, ...$ (assuming our instance is symmetric).:

$$\mathcal{H} = \begin{array}{c|ccccccc} & A & B & C & D & E & F & G \\ A & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ B & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ C & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ D & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ E & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ F & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ G & 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

We then repeatedly do this and after several iterations we might have ended up with the following H :

$$\mathcal{H} = \begin{array}{c|ccccccc} & A & B & C & D & E & F & G \\ A & 0 & 10 & 50 & 140 & 140 & 50 & 10 \\ B & 10 & 0 & 10 & 50 & 140 & 140 & 50 \\ C & 50 & 10 & 0 & 10 & 50 & 140 & 140 \\ D & 140 & 50 & 10 & 0 & 10 & 50 & 140 \\ E & 140 & 140 & 50 & 10 & 0 & 10 & 50 \\ F & 50 & 140 & 140 & 50 & 10 & 0 & 10 \\ G & 10 & 50 & 140 & 140 & 50 & 10 & 0 \end{array}$$

We then convert each of these rows and columns to percentages:

$$\mathcal{H}_{\%} = \begin{array}{c|ccccccc} & A & B & C & D & E & F & G \\ A & 0 & 2.5 & 12.5 & 35 & 35 & 12.5 & 2.5 \\ B & 2.5 & 0 & 2.5 & 12.5 & 35 & 35 & 12.5 \\ C & 12.5 & 2.5 & 0 & 2.5 & 12.5 & 35 & 35 \\ D & 35 & 12.5 & 2.5 & 0 & 2.5 & 12.5 & 35 \\ E & 35 & 35 & 12.5 & 2.5 & 0 & 2.5 & 12.5 \\ F & 12.5 & 35 & 35 & 12.5 & 2.5 & 0 & 2.5 \\ G & 2.5 & 12.5 & 35 & 35 & 12.5 & 2.5 & 0 \end{array}$$

We now have a measure of which edges are common (and assuming our algorithm is constructing good solutions, good edges). We can now introduce the idea of hot spots - building blocks with a use percentage (ie. value in $\mathcal{H}_{\%}$) of above q (where q is user defined) are hot spots. We define another constant, κ and run local searches on solutions that have no more than $\kappa\%$ of building blocks that aren't hot spots.

2.2. Ant Colony Optimization

Ant Colony Optimization is similar to Bee Colony Optimization, utilising the same idea of solution construction and daemon action phases, but instead of waggle dance phases there is a central pheromone matrix which is updated after each iteration and directly influences the solution construction step of the algorithm. The transition function is generally defined as

$$P_{i,j} = \frac{\tau_{i,j}^{\alpha}\eta_{i,j}^{\beta}}{\sum_{j \in A_{i,n}} \tau_{i,j}^{\alpha}\eta_{i,j}^{\beta}} \tag{2.2}$$

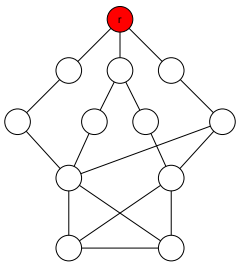
where $\tau_{i,j}$ is the pheromone value stored between nodes i and j, and η is some problem specific value. The dance phases from Bee Colony Optimization are replaced by a single pheromone update phase after the Daemon Actions phase. In this phase, pheromone is applied to each edge used in any solution, with more pheromone added to edges used in high quality solutions. After the application, each pheromone value is multipled by a constant $0 < \rho < 1$, which helps avoid algorithm stagnation, and reduces the effect of results found in earlier iterations, which will usually be poorer than later results.

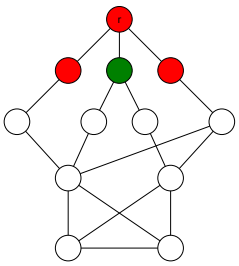
2.3. FireFighting

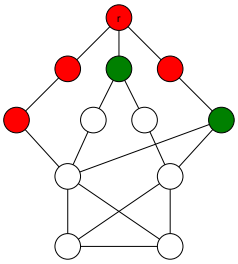
We can informally define the FireFighting problem as follows:

At time zero (time proceeds in discrete steps), a fire breaks out at some vertex (or vertices). During each discrete time step a firefighter can be placed at any vertex that is not burning (and so this vertex is now defended). Then fire spreads from any vertex that is on fire to every neighbouring vertex that is not defended. This continues until the fire can no longer spread (ie. it is either contained or all unsaved

vertices are on fire). The aim is to place the firefighters to limit the spread of fire as much as possible. Figure 2.1 contains an example of the FireFighting problem and an example solution.

Initial graph: 

After 1 timestep: 

After 2 timesteps: 

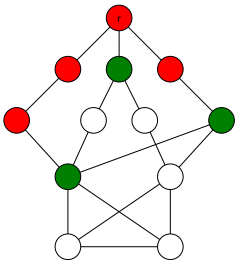
After 3 timesteps and completed graph: 

Figure 2.1.: An example of FireFighting on a small graph. Red nodes are nodes on fire and green nodes are nodes which have a firefighter placed on them.

To formally define the problem let us first define a strategy for the FireFighting problem. A strategy is a finite sequence of vertices, $d_1, d_2, ..., d_t$ of length t where vertex d_i is not on fire at time i. Once we have a strategy, we can easily verify if it is a valid solution to the FireFighting problem instance by stepping through the process as described in the informal definition and placing a firefighter on each vertex in the strategy in order. Once we complete this process we can simply count the number of vertices burnt or unburnt to determine if the solution meets the criteria.This leads

us to define the FireFighting problem:

FireFighting
Instance: An undirected graph $G = (V, E)$, a vertex $r \in V$ and an integer $k \geq 1$.
Problem: Is there a strategy for G for which at most k vertices burn, where initially only r is on fire?

FireFighting is known to be **NP**-complete on general graphs. We can easily modify the problem to create an optimization problem that will also be **NP**-hard as follows:

OPT-FIREFIGHTER
Instance: An undirected graph $G = (V, E)$ and a vertex $r \in V$.
Problem: What is the a strategy for G that minimizes the number of burnt vertices, where initially only r is on fire?

It is obvious that OPT-FIREFIGHTER is **NP**-hard; if we can solve OPT-FIREFIGHTER in polynomial time, we can solve the decision problem FireFighting in polynomial time (by solving OPT-FIREFIGHTER and find a value of x for the least possible number of vertices burning, we know the answer is YES for all $k \geq x$, and NO for all $k < x$.) and so we would prove $\mathbf{P} = \mathbf{NP}$. As we assume $\mathbf{P} \neq \mathbf{NP}$, OPT-FIREFIGHTER is **NP**-hard.

Previous Work on FireFighting

Most previous work on FireFighting has studied the complexity of the problem on a variety of graph classes, and more recently its parameterized complexity.

Perhaps the most important piece of work, at least with regards to this thesis, was the proof of **NP**-completeness of FireFighting on bipartite graphs ([FM09]). The reduction was from EXACT COVER BY 3-SETS (X3C), a known **NP**-complete problem, to FireFighting by constructing a bipartite graph. Another important

result from the survey is that FireFighting can be solved in polynomial time when applied to binary trees.

[KM10] contains a proof of **NP**-completeness for FireFighting on cubic graphs by reduction from 3-NN SAT (Not All Equal 3-SAT without negated literals).

It has been shown that FFP is **NP**-hard on trees of maximum degree 3 ([CFvL12]), and so FireFighting is **NP**-complete for general trees . Additionally, it has been shown that for any optimal solution for FireFighting on trees, $s_1, s_2, ..., s_n$, any choice in the solution s_i will be adjacent to the fire at time i, and thus has depth i from the root ([MW03]).

2.4. The Quadratic Assignment Problem

The Quadratic Assignment Problem (QAP) is believed to be one of the most difficult optimization problems studied, with few problems over size 50 solved exactly. The problem is defined as follows: we have 2 sets, a set of n locations L with distance matrix l, and a set of n facilities F with flow matrix f, where $l_{i,j}$ is the distance between location i and location j, and $f_{x,y}$ is the flow between facilities x and y (for example if a hospital were modeled as this problem, the flow could be patient travel between departments). Our aim is to find an assignment of facilities to locations, ϕ that minimizes the equation:

$$z_{QAP} = min \sum_{i=1}^{n} \sum_{j=1}^{n} l_{i,j} f_{\phi(i),\phi(j)} \tag{2.3}$$

For the hospital example, this would equate to minimizing the overall travel for patients between departments.

The problem tends to be attacked in one of two ways. The first involves lower bound ascent procedures, which compute lower bounds then attempt to ascend systematically to try to find an optimum solution. A large amount of work in the field was initially based on computing lower bounds for problems, and one such example is the Gilmore-Lawler Bound [Gil62], so these ascent procedures have many bounds to work with. The second is a standard optimization approach, using genetic algorithms, tabu search or Ant Colony Optimization (which is a broad title given to a family of similar algorithms). There has also been some work on optimal algorithms, many of which use a branch and bound algorithm, such as the exact algorithm in [MDBS98].

Previous Optimization Work

For problems in QAPLIB ([QAP]) that haven't been solved to optimality, the vast majority of best known results were found by Tabu Search (most commonly the Robust Tabu Search [Tai91]). Other good algorithms on QAPLIB instances have been genetic algorithms ([FJF93]) and Ant systems ([MC99]). Other algorithms have produced similar results, such as Approximate Nondeterministic Tree Search ([MDBS98]) and simulated annealing ([BR84]).

Tabu (or sometimes Taboo) Search for the QAP is based on a simple switch procedure. It has been noted that if we have a solution μ obtained from solution π by a 2-opt move switching two assigned facilities (or locations) r and s we can quickly compute the value of solution μ if we know the value of solution π, which speeds up the process dramatically over recalculating the value of solution μ. In Robust Tabu Search (Ro-TS), the algorithm has 2 defined parameters t and u, and allows moves at iteration a (regardless of if they improve the solution) if they satisfy the following rules (taken from [Tai95]):

a) If $a > t$, and a facility i hasn't been at location x in the last t iterations, the available set of moves must all place facility i at location j.

b) If during the last u iterations facility j was placed at location y and facility k at location z then the available set of moves contains no solutions which contain j placed at y and k placed at z unless the solution is the better than the best solution found so far.

The algorithm creates a set of allowed moves from iteration k, Π^k, and selects the move with the best value in the move set to be the new move π^{k+1}. The values of u and t depends on the instance being solved (ideally) and promote variation of the move set.

Ant system for QAP is the standard Ant system model with a 2-dimensional matrix (an explanation of how Ant System works generally is discussed in section 3.4). In [MC99], they used the Gilmore-Lawler Bound as the measure of how good an assignment is. ANTS ([MDBS98]) used a simpler lower bound, whereas Min-Max Ant System ([Stü97]) used only the pheromone, but mixed with a stronger local search (using Ro-TS instead of a simple 2-opt mechanism). The results for all 3 were good, however Ant System was slower due to the time required to calculate the Gilmore-Lawler bound.

3. Bee Colony Optimization for the FireFighting problem

3.1. Graph Centrality

Before we discuss the algorithm for the FireFighting problem, we need to introduce the concept of graph centrality.

Centrality

Centrality is a measure of how "central" a node is to a graph's structure. A simple example is in a star graph. The node with high degree would often have a high centrality value, whereas the leaves would all have lower centrality. Also in node symmetric graphs, isomorphic nodes should have the same centrality measures. A variety of centrality measures are highlighted below.

Degree Centrality

The degree centrality of a node in a (di)graph is simply the (out-)degree of the node. The runtime to calculate this centrality for a node takes $O(n)$ time to compute if the

graph is stored as an adjacency matrix, but $O(1)$ time to compute from adjacency sets (it is simply the size of the adjacency set).

PageRank Centrality

The PageRank centrality of a node is a refinement of degree centrality, and is based upon eigenvector centrality and is similar to PageRank ([PBMW99]), except we do not use a transportation matrix as our graph is undirected and connected. Eigenvector centrality takes into account that not all links in a graph are equal in importance ([New08]) and so attempts to weight nodes connected to more important edges higher than other nodes. PageRank centrality is calculated from an adjacency matrix, so if our graph is stored as adjacency sets, we need to convert it to an adjacency matrix, which takes $O(n^2)$ time. Once we have an adjacency matrix, we normalize each column so that they add to 1 (ie. for a vertex x, the values in column x in our matrix are either 0 if a link doesn't exist or $\frac{1}{deg(x)}$ if a link exists). For example if our graph is:

$$G = \begin{matrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 \end{matrix}$$

(G is an $n \times n$ matrix) after normalization we get:

$$G' = \begin{matrix} 0 & 1 & 0 & \frac{1}{2} & 0 \\ \frac{1}{3} & 0 & 1 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 0 \\ \frac{1}{3} & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & \frac{1}{2} & 0 \end{matrix}$$

We then create our initial column vector which contains our centrality values, and as our adjacency matrix is normalized, the vector of $\frac{1}{n}$s is a good initial estimate. We then use the power iteration to repeatedly compute vectors until the values in this vector converge, which usually takes no more than 50 iterations. The power iteration works by using the following formula, $\nu^{i+1} = \frac{G' \cdot \nu^i}{sum(G' \cdot \nu^i)}$, where ν^i is the vector found at iteration i and $sum(G' \cdot \nu^i)$ is the sum of the elements of $G' \cdot \nu^i$, thus the values in ν^{i+1} always sum to 1. We observed the vector converging similarly to PageRank despite the lack of transportation matrix due to the graph being undirected and connected. This results in a runtime of $O(n^2)$ time on adjacency sets, or $O(n)$ on adjacency matrices. The PageRank centrality of node n is then the n^{th} value of the final vector computed.

Betweeness Centrality

Betweeness centrality is a measure of how much load there is on a node if traffic is passing through the network. It is calculated as the number of shortest paths between all pairs of nodes z, y that a node x lies on, ie. the number of paths $y, ..., x, ..., z$ ($x = y$ is allowed, but $x \neq z$) that are the shortest paths between y and z. We can easily find the shortest paths between a node z and all other nodes in $O(|V|^2)$ time, so betweeness centrality can be calculated in $O(|V|^3)$ time. [Bra01] created an algorithm to compute this in $O(|V||E|)$ time on unweighted graphs, which for sparse graphs is likely to be better than our simple $O(|V|^3)$ bound. This improvement is attained by using the *Bellman criterion,* which states: A vertex $v \in V$ lies on a shortest path between vertices $s, t \in V$ iff $d_G(s,t) = d_G(s,v) + d_G(v,t)$, where $d_G(s,t)$ is the shortest distance between s and t.

Closeness Centrality

Closeness centrality is a simple measure of closeness ie. how close a node is to all other nodes. First, define the farness of a node s as the sum of its distances to all other nodes. Then closeness is the inverse of this. To calculate closeness, we first use breadth first search algorithms, one starting at each node, to compute the farness of each node. Breadth first search takes $O(|V| + |E|)$ time per search, so the total time for this stage is $O(|V| \cdot [|V| + |E|])$. Once this is computed, we can simply find the reciprocal of this value to obtain the closeness centrality of a node.

3.2. The Updated Algorithm

As noted in section 2, BCO requires some way to measure how good a node may be if selected to be added to a solution during the constructive phase ($\eta_{i,j}$ in equation 2.1). For FireFighting there may be a variety of effective parameters, however most of the ones we tested were related to either centrality or distance from the root (or a combination of both). Centrality seems a good measure, as its definition implies nodes with high centrality (and thus high probability to be selected in our solutions) will significantly affect connectedness of the graph, so saving such nodes should hopefully either increase the diameter of the graph (giving us more selections) or act as a vertex cut, completely isolating a section of the graph from the fire. In section 2 we reviewed 4 different types of centrality, all of which were tested in the algorithm. We also introduced a variation on betweeness centrality for this purpose which we have named single betweenness.

Betweeness centrality computes centrality of a node amongst the whole graph, so the obvious change to make for the FireFighting problem is to compute a version of betweeness that considers only paths from the root.

Single Betweenness was computed using a simple algorithm, where the graph is stored as adjacency matrix.

- First, starting at the root, we run breadth first search and find the depths of all nodes. During this search, when we discover a node of depth d we store all of its parents with depth $d-1$. This process gives us all shortest paths in the graph between r and any other node v.
- Secondly, for each node we count the number of shortest paths it lies on between the root and all other nodes, and this gives us the centrality value of each node.

We also introduced some simple upper and lower bounds on the number of nodes that could be saved when selecting a node (discussed in section 3.3). However, due to the "turn-based" nature of the FireFighting problem, very little information about the future fire spread can be obtained by saving one node, so neither bound is tight (whereas for a problem like QAP these bounds can be fairly tight and provide a lot of information for constructive algorithms for some instances).

Due to the minimal difference between optimal solutions and any solution found by the algorithm for most instances, all bees have a 50% chance of using their previously found solution and an equal chance to follow another bees solution (providing any bees are advertising solutions) during the "Follow *Waggle Dances*" stage of the algorithm. The time that bees advertised solutions was set to be a number of iterations equal to twice the value of the solution (the value of the solution is the number of saved vertices) found for the same reason. If no bees had danced in 50 iterations the memory adjustment policy was applied to all bees. These values were based upon experimental observation.

From an experimental point of view, it can be easier to think of the FireFighting problem in a new way. During each time step we do the following: 1) Select a vertex

v and remove v and all incident edges from the graph. 2) Take our root, r and merge all adjacent nodes to create a new "supernode", r', which becomes adjacent to all nodes that were adjacent to the merged nodes. As such, we create a new graph $G' = G \backslash (v \cup N(r))$ with root r', where $N(r)$ is the set of neighbours of r. This ensures only our root node is ever on fire. We can then repeat this process until $|N(r)| = 0$ in which case the fire can no longer spread and we can stop.

3.3. Local Search Algorithms

Due to the "turn-based" nature of the FireFighting problem, a standard 2-opt algorithm will almost certainly fail to find improvement due to the strategies being invalid (as no longer saving a vertex could change the way the fire spreads). 2-opt for FireFighting simply consists of selecting a node in our strategy, and replacing it with a node not otherwise selected if that change improves our strategy. Instead, we placed restrictions on our 2-opt decisions, to either save a vertex at the same depth or to save a vertex with depth one less, as well as a random choice. Figures 3.1 and 3.2 show how these algorithms could work on a simple problem.

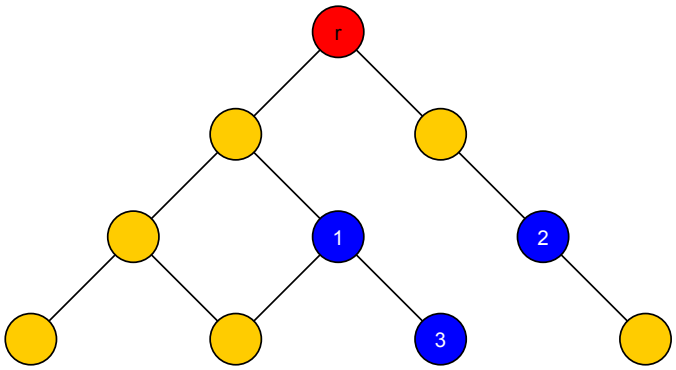
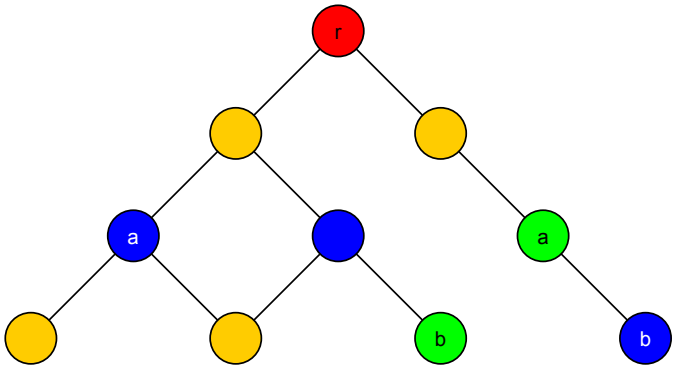
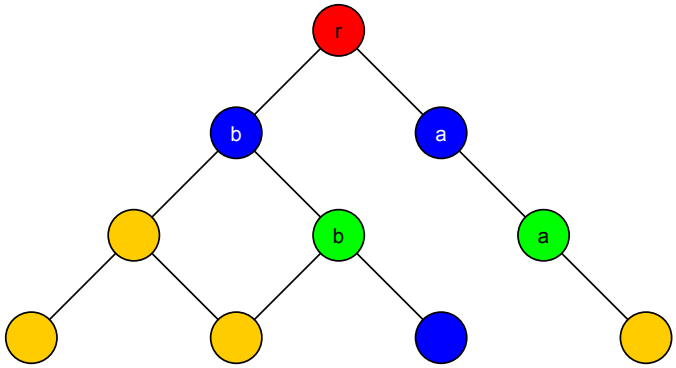


Figure 3.1.: A simple graph with an example strategy shown. Vertices in blue were selected as the solution before local search.

a)





b)

Figure 3.2.: Results of 2 local search algorithms. a) shows the result of our same depth restriction, b) the result of save vertices with depth one less. Vertices in green are vertices that were saved but are no longer after our local search. Clearly b) is not a valid strategy, however these can sometimes be useful, as shown in figure 3.3.

One further idea we used to attempt to improve these algorithms was not to immediately discard a move that makes the strategy invalid. A situation could occur as highlighted in figure 3.3, where we use an invalid strategy, before moving back to a valid, better strategy.

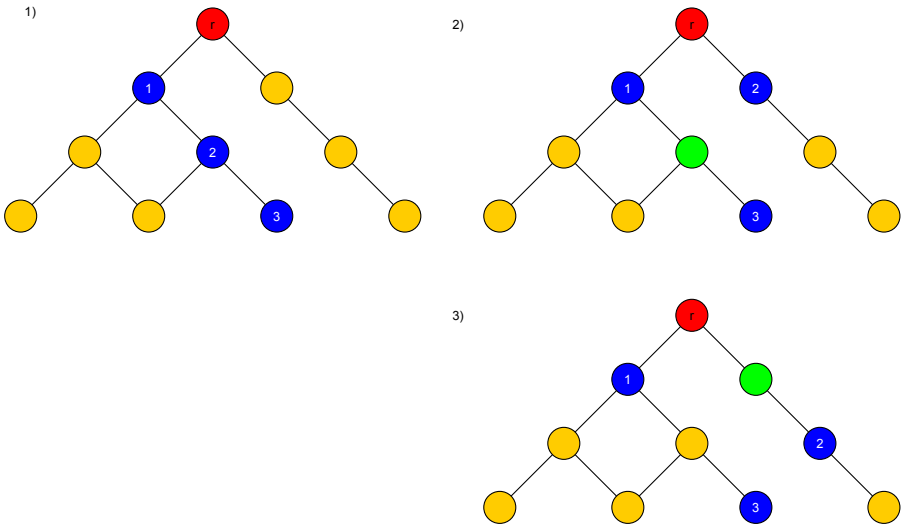


Figure 3.3.: An example of improvement via invalid strategies during random local search.

However, we returned to the last valid strategy if we didn't return to a strategy within a small number of iterations of local search, as the repeated illegal moves made the chance of moving back to a valid strategy much lower.

3.4. Greedy Algorithms

We created 2 simple algorithms that could be used as comparison algorithms with our Bee Colony Algorithm. The first algorithm is based on selecting the node at each timestep which gives the largest increase of the number number of vertices that can no longer be burnt, L_v. This is easy to compute for each $v \in V$ as $L_v = |V| - |U| + D(G \backslash v, r) - 1$ where $U \subseteq (V \backslash v)$ is the set of vertices reachable from the root node, r, once v is removed and $D(G \backslash v, r)$ is the greatest distance of any node from r in $G \backslash v$.

The second algorithm is a little less obvious. We know we cannot select more than 2 vertices with depth less than or equal to 2, and we can then define for each $v \in V$,

$U_v = |V| - \mathcal{Z}(G \backslash v) + 1$, where $\mathcal{Z}(G \backslash v)$ is the number of nodes in $G \backslash v$ with depth 2 or less. In the greedy algorithm we then select the vertex in the graph with maximum value of U_v. An obvious downside of this method is nodes with depth > 1 will always have a poor value as they will not affect the size of $\mathcal{Z}(G \backslash u)$, whereas nodes close to the root have the potential to increase the depth of other nearby nodes (since the direct paths to them from the fire are removed).

The implemented algorithms were simple. For the first method, we first calculate L_v for all $v \in V$. Then, we set the node picked at time 1 to the node v with highest value of L_v, combine the adjacency set of r with those of all of its neighbours (excluding v if v is a neighbour of r) and set $G_1 = G \backslash (N(r) \cup v)$ (ie. G_1 has a root node, r that is adjacent to all nodes with depth 2 in G excluding those with only reachable via v from r). We then repeat this calculation and picking until the fire no longer spreads, and our strategy is the nodes picked at each time step during the algorithm.

3.5. Ant Colony Optimization Comparison Algorithm

We also implemented a simple Ant system algorithm for the FireFighting problem. We used a 1-dimensional pheromone matrix, Π where π_a represented the 'quality' of selecting node a calculated from solutions computed so far during the algorithm. The pheromone matrix was initially set so every value was $\frac{1}{n}$, where n is the number of vertices in the graph.

The transition rule was based on rules for both TSP and QAP that have shown Ant Colony Optimization algorithms are capable of finding good solutions ([DS04],[Stü97]) without access to a node quality measure other than pheromone values. Let us denote P_{ij} as the probability of choosing node j at timestep i and η_{ij} as 1 if choosing node j at timestep i of the solution is a legal move, and 0 otherwise. This results in

a transition rule described by:

$$P_{ij} = \frac{\pi_j \eta_{ij}}{\sum_{\forall j \in G} \pi_j \eta_{ij}} \tag{3.1}$$

The pheromones were updated in each iteration. We first calculated a value τ_i for each node i, where for each time i appeared in a solution S, we added $\frac{1}{val(S)}$ to τ_i, where $val(S)$ was the number of nodes saved by the solution. Once these values were calculated for all nodes, we updated the pheromone matrix so for each node i, $\pi_i^{k+1} = 0.95 \cdot \pi_i^k + 0.05 \cdot \tau_i$, where π_i^k is the pheromone value of node i at time k.

3.6. Problem Instances

As FireFighting is known to be polynomial-time solvable for binary trees ([MW03]), and in general for trees the optimal solution will save a node at depths $1, 2, 3...d$,we ran the algorithm on graphs that contained at least 1 cycle. We decided not to use trees as our algorithm was to be as general as possible - however it would be easy to add something to recognize a tree input and restrict to save nodes at depths $1, 2, 3...d$. Problems were randomly generated by constructing a spanning tree over n nodes, and then adding m edges to the spanning tree until the graph was constructed. Some instances are listed in appendix 1 - the format is adjacency lists with each list on a separate line. Most of the instances are quite sparse (as these have larger solution values and optimal solution sizes) and a few have 1 vertex optimal solutions (ie. the graph has a vertex cut of size 1 at depth 1).

3.7. Experimental Results

All experiments were run on a Intel Pentium G840 CPU running at 2.8 GHz using only 1 core. We set $\lambda = 0.95$, $\alpha = 1.0$, $\beta = 1.0$. The BCO algorithms were run 10 times on each instance, and the worst and best results found by the algorithm with each centrality measure tested are presented in the table below.

Each centrality measure was tested with all 3 local searches, however the results are not shown here as they did not improve the solutions found.

Problem Size		Minimum/Median/Maximum number of nodes saved (η set to column title)						Nodes Saved		
Nodes	Edges	None	Degree	Closeness	Single Betweeness	Betweeness	PageRank Betweeness	L_v	U_v	Ant Colony Optimization
10	17	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	3	**4**	**4**
14	26	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	**4/4/4**	**4**	**4**	3
27	35	10/15/**16**	15/**16/16**	**16/16/16**	**16/16/16**	15/**16/16**	**16/16/16**	15	15	10
33	59	9/11/**13**	11/**13/13**	9/11/**13**	**13/13/13**	**13/13/13**	11/**13/13**	6	9	4
42	61	13/**40/40**	**40/40/40**	**40/40/40**	**40/40/40**	**40/40/40**	**40/40/40**	14	**40**	9
46	56	**44/44/44**	**44/44/44**	**44/44/44**	**44/44/44**	**44/44/44**	**44/44/44**	**44**	**44**	14
48	74	**47/47/47**	**47/47/47**	**47/47/47**	**47/47/47**	**47/47/47**	**47/47/47**	**47**	**47**	**47**
55	104	9/10/13	13/13/**15**	12/13/**15**	14/**15/15**	13/14/**15**	13/14/**15**	11	12	7
60	81	**59/59/59**	**59/59/59**	**59/59/59**	**59/59/59**	**59/59/59**	**59/59/59**	**59**	**59**	11
61	78	**57/57/57**	**57/57/57**	**57/57/57**	**57/57/57**	**57/57/57**	**57/57/57**	**57**	**57**	14
62	80	**61/61/61**	**61/61/61**	**61/61/61**	**61/61/61**	**61/61/61**	**61/61/61**	**61**	**61**	16
63	106	11/12/15	13/13/15	11/12/15	13/16/**17**	14/15/**17**	13/14/14	10	12	8
72	135	8/8/8	8/8/**10**	8/8/8	9/**10/10**	9/9/**10**	8/8/9	8	7	5
84	116	18/**83/83**	**83/83/83**	**83/83/83**	**83/83/83**	**83/83/83**	**83/83/83**	**83**	**83**	12
92	138	17/19/21	19/19/20	16/17/19	21/21/**22**	20/21/21	18/19/20	14	18	10
102	175	13/15/**16**	14/15/**16**	12/13/15	15/**16/16**	**16/16/16**	14/15/**16**	**16**	13	8
113	199	14/16/17	18/19/20	14/15/16	19/20/**21**	19/20/**21**	16/18/18	16	20	7
122	217	11/12/14	15/17/17	13/13/15	15/17/18	16/17/**19**	15/16/16	14	12	6
129	182	16/16/**127**	126/**127/127**	**127/127/127**	**127/127/127**	126/**127/127**	126/**127/127**	24	**127**	16
139	272	12/13/14	13/14/14	11/12/13	14/15/**16**	14/14/15	13/13/14	15	13	6
143	183	26/27/**141**	**141/141/141**	24/**141/141**	**141/141/141**	**141/141/141**	**141/141/141**	30	**141**	11
145	269	10/11/**13**	11/12/**13**	10/11/11	12/**13/13**	**13/13/13**	11/11/12	11	9	6
147	209	13/14/21	20/23/25	16/17/21	25/27/**28**	26/27/**28**	20/21/22	20	17	13
157	217	20/21/30	32/34/36	21/24/29	35/38/39	38/39/40	29/30/33	35	**41**	16
400	607	19/19/21	23/24/25	18/20/23	**397/397/397**	25/27/29	21/22/26	29	**397**	14
407	770	11/12/13	11/12/**14**	11/11/12	13/13/13	13/**14/14**	11/12/13	**14**	11	7
423	806	9/12/14	12/13/14	12/12/14	13/14/15	14/14/15	12/13/13	**23**	11	9
800	1178	21/23/27	25/26/28	22/23/24	27/30/33	29/31/33	24/25/28	**35**	24	13

Table 3.1.: Algorithmic results on randomly generated FireFighting instances.

The remaining examples are on a selection of problems which had good variation on results found.

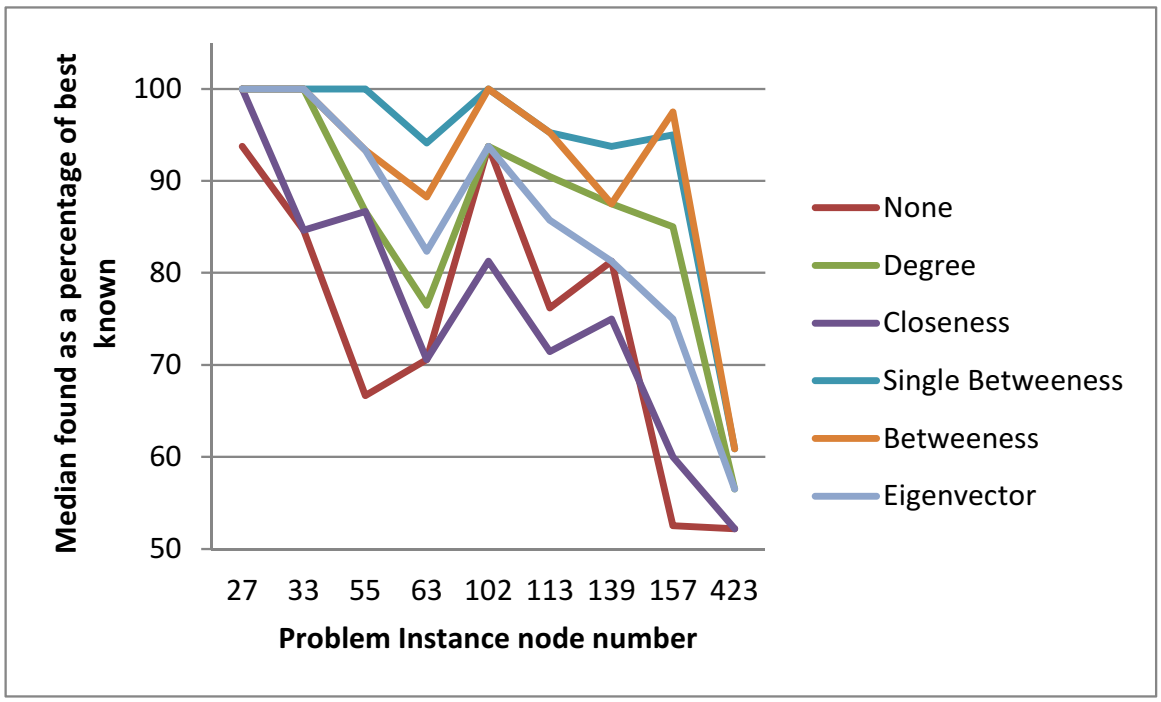


Figure 3.4.: A graph that shows the median result found by each Bee Colony Algorithm as a percentage of the best known on selected instances. These instances are available in the appendix, excluding the size 423 instance.

Nodes	Edges	Mean iterations to find best result	
		Betweeness	Single Betweeness
27	35	71.2	10.8
33	59	46	67.2
55	104	211	87.6
63	106	184.2	357.8
102	175	118.8	152.2
113	199	146	178.6
139	272	69.6	168.2
157	217	209.6	308
423	806	87.2	169.4

Table 3.2.: Iterations required to find best solutions on a selection of FireFighting instances

3.8. Conclusions and Further Work

In this chapter we have introduced the FireFighting problem as an optimization problem, and attacked it with both Bee Colony and Ant Colony Optimization algorithms. We have also attempted to produce some local search algorithms for the problems based on methods known to be effective for other optimization problems. Finally we introduced simple greedy algorithms for the problem.

The results of the algorithm show that the goodness measure does have a substantial impact on the results found for the Bee Colony Optimization algorithms, as the results with no centrality are on average worse than those with a centrality measure. The results showed that either betweeness centrality or single betweeness centrality gives the best results of any centrality measure on almost all problems under size 400, and generally obtains better results than the greedy algorithms on these problems. On large problems (400+ nodes), the algorithm seems to perform worse, possibly due to the methods we used. These problems have higher diameter from the root (9-11 on average as opposed to 6-8 for problems of size 100-200 or 3-5 on problems of under 100 nodes), and our algorithms didn't explicitly attempt to select nodes near the root, though it would be expected single betweeness will favour such nodes. However, on most of these larger problems single betweeness appears to perform worse than betweeness, so it seems further investigation is needed, as neither perform as well as a greedy algorithm using L_v to build its solutions. Another idea to attempt to improve such measures could be to reintroduce a variant on PageRank's transportation matrix for the PageRank centrality measure, except set this transportation vector to exclusively favour links connected to the root very slightly. So for example, our power iteration now calculates $\nu^{i+1} = \frac{\alpha \cdot (G^N \cdot \nu^i) + (1-\alpha) \cdot x}{sum(\alpha(G^N \cdot \nu^i) + (1-\alpha) \cdot x)}$ where α is some constant between 0 and 1, and x is our transportation vector, which could be contain values of $\frac{1}{d}$ where d is the depth of each node from the root, normalised

so the sum of the elements of x is 1, and x is a column vector.

The Bee Colony Optimization algorithms often find a good solution in the first few iterations and will stagnate for a period, and then improve again after a large number more iterations. This improvement appears after long periods waiting for the memory weakening criteria to be met, and so it seems likely this runtime could perhaps be improved by reducing (and normalising) dance duration, so larger problems don't have solutions advertised for as many iterations, and by reducing the number of iterations before applying the memory weakening criteria from 50 to around 10-20.

The results of the greedy algorithm seem to suggest a change in effectiveness of the algorithms as the size of the problem instances increase. The results indicate that the U_v algorithm performs better on smaller problems, or on problems where the optimal solution size is small. On larger problems, the L_v algorithm seems more effective, however more experimentation on large problems would be needed to confirm this.

The Ant Colony Optimization algorithm performed worse than was expected, though the lack of a strong local search algorithm perhaps hindered it more than the Bee Colony algorithms. Swarm Intelligence algorithms are strongest when you can combine their constructive search methods with many short runs of a strong local search algorithm. Bee Colony found reasonable solutions without the local search algorithm, likely due to the fact the centrality measures employed seem to be effective indicators of good selections in the problem, whereas Ant Colony was created to work without these measures. The ACO algorithm performed worse than BCO without any measure on most problems. It is reasonable to assume that if we added these centrality measure to Ant Colony it could perform as well or better than Bee Colony for the FireFighting problem, and this is a potential area for further work.

One issue with the centrality measures tested was due to the way the graph was morphed after each selection, these values couldn't be recalculated due to the expense of computing them. An area to investigate with these measures is finding a cheaper way to recalculate these based upon changes to the graph, ie. if we remove a vertex and know previous information about centrality values, can we cheaply compute changes to centrality values, or even approximate these changes?

Another important development would be a local search algorithm that is effective at improving solutions. None of the ideas we tested in this study worked, so it seems likely a different style of local search than a simple switch might be required.

4. Bee Colony Optimization for the Quadratic Assignment Problem

4.1. The Quadratic Assignment Problem

The QAP is an optimization problem that can be defined as follows. We have two sets, a set of n locations L with distance matrix l and a set of n facilities F with flow matrix f. Our aim is to find an assignment ϕ that minimizes the equation:

$$z_{QAP} = min \sum_{i=1}^{n} \sum_{j=1}^{n} l_{i,j} f_{\phi(i),\phi(j)}$$

This definition is discussed in more detail in Section 2.4.

4.2. BCO for the QAP

The BCO algorithm we have implemented does not use the Gilmore-Lawler bound ([Gil62]) due to the computational time required, despite this being commonly used in many constructive algorithms for the QAP. Instead we used a much simpler

measure of η. The general structure of the algorithm is discussed in detail in section 2.1, and we highlight changes to the algorithm to use it on the QAP in this section.

In the solution construction step, each individual creates a solution probabilistically according to Equation 4.1. $P_{i,j}$ is the probability that element j is added to the solution as the n^{th} element where element i was the $(n-1)^{th}$ element. We also let θ denote the element that followed i in S^*, λ is a predefined constant in the range $[0, 1]$, and $A_{i,n}$ is the set of available choices from i (the choice made at time $n-1$ during this solution construction phase by this bee, so in QAP $A_{i,n}$ is every unassigned facility (or location) remaining) at time n. At time 0 we say the bee is at the hive, and so the set $A_{i,n}$ is the set of all possible choices. We then define $P_{i,j}$ as:

$$P_{i,j} = \frac{[p_{i,j}]^\alpha [\eta_{i,j}]^\beta}{\sum_{j \in A_{i,n}} [p_{i,j}]^\alpha [\eta_{i,j}]^\beta} \tag{4.1}$$

where η is some problem specific value and $p_{i,j} = \begin{cases} 0, & j \notin A_{i,n}, \\ \lambda, & j = \theta, \theta \in A_{i,n}, \\ \frac{1-\lambda}{|A_{i,n}|-1}, & j \neq \theta, \theta \in A_{i,n} \\ \frac{1}{|A_{i,n}|}, & \theta \notin A_{i,n}. \end{cases}$,

We use the values of α, β to encourage bees to either follow previous iteration results more closely (by increasing the value of $\frac{\alpha}{\beta}$) or to pay more attention to the problem specific values (by decreasing the value of $\frac{\alpha}{\beta}$).

The first thing to note for QAP is that we have to either assign either facilities to locations or locations to facilities. There is no strict way to say which is better, our results showed that for some problems facilities to locations works best, and

for others, the opposite. Let us assume we are assigning facilities to locations, we first need to somehow pick an order of assignment for the locations. Most past work on constructive algorithms has shown that having some rule to decide this is much better than doing it at random. I used a simple idea known as location/flow potential: Assign to the locations (or facilities in the reverse) with the highest centrality first (ie. those locations i such that $\sum_{j \in L, j \neq i} l_{i,j}$ is smallest are assigned to first).

The next step was to decide on an easily computable value for η. The value of η in this algorithm uses a a simple idea based on flow and location potentials. Let us define 2 subsets of L and F, the already assigned locations and facilities, L_1 and F_1 , and the unassigned locations and facilities L_2 and F_2. Now we can define a new Π_i to be the flow potential for some facility (or location) $i \in F_2$(or L_2) to be

$$\Pi_i = \sum_{k \in F_2, k \neq i} f_{i,k}$$

Then we can define:

$$\eta_{i,j} = \sum_{x \in L_1} (l_{x,j} f_{\phi(x),i} + l_{j,x} f_{i,\phi(x)}) + \Pi_i \tag{4.2}$$

where i is our facility we are calculating for and j is the next location to be assigned to.

4.3. Local Search Algorithms

We tested a few different local search techniques with the BCO algorithm. The first is a simple 2-opt switch, where we take any pair of assignments in the solution and

switch the assigned nodes (for example $[1, 3, 4, 2]$ could switch 1 and 4 to become $[4, 3, 1, 2]$) if the switch would improve the solution. We used both methods of doing this switch, either finding an improvement and immediately switching (first-find improvement) or calculating all switches for the whole neighbourhood and making the best switch (best-find improvement)

The other local search technique used was the Robust-Tabu Search algorithms (RoTS) ([Tai91]) which I have converted to Java.

4.4. Experimental Results

We ran each algorithm 5 times on each problem instance, and the best results are shown in the tables below. Problems where a * follows the given optimal are problems that have not been solved to optimality and so the best known result has been shown instead where possible. The results with Bee Colony Optimization use Robust-Tabu Search as the local search algorithm. This algorithm performed significantly better than the simple 2-opt local search algorithm. We used runs of 300 iterations on each solution found.

All 3 algorithms were implemented in Java 7. The Fast Ant algorithm is found in [Tai98] and uses Taillard's implementation converted to Java. Both the Ant and Bee algorithms run until 1000 iterations have been performed without improving the solution. The Robust-Tabu Search runs 100000 iterations. The t_{best} times shown are the quickest times the algorithm required to find its best solution.

Problem Instance	Optimal	BCO		Fast Ant		Ro-TS	
		Result	t_{best}	Result	t_{best}	Result	t_{best}
bur26a	5426670	**5426670**	0.1	5431633	0.1	**5426670**	0.2
bur26b	3817852	**3817852**	0.3	3824777	0.2	**3817852**	0.1
bur26c	5426795	**5426795**	0.1	5427564	0.1	**5426795**	0.2
bur26d	3821225	**3821225**	0.1	3822307	0.2	**3821225**	0.2
bur26e	5386879	**5386879**	0.1	5387620	0.1	**5386879**	0.0
bur26f	3782044	**3782044**	0.1	3782713	0.3	**3782044**	0.2
bur26g	10117172	**10117172**	0.1	10119734	0.3	**10117172**	0.1
bur26h	7098658	**7098658**	0.0	7098905	0.2	**7098658**	0.2
chr12a	9552	**9552**	0.0	11514	0.0	**9552**	0.0
chr12b	9742	**9742**	0.0	10596	0.0	**9742**	0.0
chr12c	11156	**11156**	0.0	11974	0.0	**11156**	0.0
chr15a	9896	**9896**	0.0	10890	0.0	**9896**	0.0
chr15b	7990	**7990**	0.0	10784	0.0	**7990**	0.0
chr15c	9504	**9504**	0.0	14082	0.0	**9504**	0.0
chr18a	11098	**11098**	0.0	17068	0.0	**11098**	0.0
chr18b	1534	**1534**	0.0	1740	0.0	**1534**	0.0
chr20a	2192	**2192**	2.3	3028	0.0	**2192**	0.6
chr20b	2298	**2298**	0.5	2880	0.1	**2298**	0.5
chr20c	14142	**14142**	0.3	17942	0.0	**14142**	0.1
chr22a	6156	**6156**	0.1	6730	0.1	**6156**	0.4
chr22b	6194	**6194**	1.7	7034	0.1	**6194**	0.4
chr25a	3796	**3796**	3.3	5594	0.2	3874	0.8
els19	17212548	**17212548**	0.0	18080088	0.0	**17212548**	0.0
esc16a	68	**68**	0.0	**68**	0.0	**68**	0.0
esc16b	292	**292**	0.0	**292**	0.0	**292**	0.0
esc16c	160	**160**	0.0	**160**	0.0	**160**	0.0
esc16d	16	**16**	0.0	**16**	0.0	**16**	0.0
esc16e	28	**28**	0.0	**30**	0.0	**28**	0.0
esc16f	0	**0**	0.0	**0**	0.0	**0**	0.0
esc16g	26	**26**	0.0	**28**	0.0	**26**	0.0
esc16h	996	**996**	0.0	**996**	0.0	**996**	0.0
esc16i	14	**14**	0.0	**14**	0.0	**14**	0.0
esc16j	8	**8**	0.0	**8**	0.0	**8**	0.0
esc32a	130	**130**	0.5	156	0.9	**130**	0.3
esc32b	168	**168**	0.0	196	0.5	**168**	0.1
esc32c	642	**642**	0.0	**642**	0.3	**642**	0.0
esc32d	200*	**200**	0.0	204	0.3	**200**	0.0
esc32e	2	**2**	0.0	**2**	0.0	**2**	0.0
esc32g	6	**6**	0.0	**6**	0.0	**6**	0.0
esc32h	438	**438**	0.0	**438**	0.6	**438**	0.1
esc64a	116	**116**	0.0	124	1.7	**116**	0.0
esc128	64	**64**	0.2	94	68.3	**64**	34.4

Problem Instance	Optimal	BCO		Fast Ant		Ro-TS	
		Result	t_{best}	Result	t_{best}	Result	t_{best}
had12	1652	**1652**	0.0	**1652**	0.0	**1652**	0.0
had14	2724	**2724**	0.0	**2724**	0.0	**2724**	0.0
had16	3720	**3720**	0.0	**3720**	0.0	**3720**	0.0
had18	5358	**5358**	0.0	5374	0.0	**5358**	0.0
had20	6922	**6922**	0.0	6928	0.1	**6922**	0.0
kra30a	88900	**88900**	0.0	91890	0.7	**88900**	0.0
kra30b	91420	**91420**	6.8	94200	0.4	**91420**	0.2
lipa20a	3683	**3683**	0.0	3781	0.0	**3683**	0.0
lipa20b	27076	**27076**	0.0	30910	0.1	**27076**	0.0
lipa30a	13178	**13178**	0.0	13444	0.6	**13178**	0.0
lipa30b	151426	**151426**	0.0	174094	0.5	**151426**	0.0
lipa40a	31538	**31538**	0.5	32044	1.4	**31538**	0.2
lipa40b	476581	**476581**	0.0	569498	1.8	**476581**	0.0
lipa50a	62093	**62093**	8.4	63173	1.2	**62093**	0.5
lipa50b	1210244	**1210244**	0.0	1463854	2.1	**1210244**	0.1
lipa60a	107218	**107218**	55.7	108833	4.3	**107218**	0.5
lipa60b	2520135	**2520135**	0.3	3111756	7.6	**2520135**	0.2
lipa70a	169755	**169755**	154.4	172209	5.0	**169755**	3.1
lipa70b	4603200	**4603200**	0.1	5743941	9.5	**4603200**	0.1
lipa80a	253195	**253195**	1933.2	256653	9.7	**254407**	2.5
lipa80b	7763962	**7763962**	0.8	9760832	17.0	**7763962**	0.0
lipa90a	360630	**360630**	784.1	365333	19.6	**360630**	22.1
lipa90b	12490441	**12490441**	3.0	15647060	26.6	**12490441**	0.4
nug12	578	**578**	0.0	586	0.0	**578**	0.0
nug14	1014	**1014**	0.0	1048	0.0	**1014**	0.0
nug15	1150	**1150**	0.0	1182	0.0	**1150**	0.0
nug16a	1610	**1610**	0.0	**1610**	0.0	**1610**	0.0
nug16b	1240	**1240**	0.0	1280	0.0	**1240**	0.0
nug17	1732	**1732**	0.0	1770	0.0	**1732**	0.0
nug18	1930	**1930**	0.0	1962	0.1	**1930**	0.0
nug20	2570	**2570**	0.0	2632	0.1	**2570**	0.0
nug21	2438	**2438**	0.0	2498	0.0	**2438**	0.0
nug22	3596	**3596**	0.0	3656	0.1	**3596**	0.0
nug24	3488	**3488**	0.0	3566	0.1	**3488**	0.0
nug25	3744	**3744**	0.0	3814	0.2	**3744**	0.0
nug27	5234	**5234**	0.0	5304	0.2	**5234**	0.0
nug28	5166	**5166**	0.0	5300	0.3	**5166**	0.0
nug30	6124	**6124**	0.0	6224	0.6	**6124**	0.2
rou12	235528	**235528**	0.0	249852	0.0	**235528**	0.0
rou15	354210	**354210**	0.0	374428	0.0	**354210**	0.0
rou20	725522	**725522**	0.1	745782	0.0	**725522**	0.2

Problem Instance	Optimal	BCO		Fast Ant		Ro-TS	
		Result	t_{best}	Result	t_{best}	Result	t_{best}
scr12	31410	**31410**	0.0	32768	0.0	**31410**	0.0
scr15	51140	**51140**	0.0	54666	0.0	**51140**	0.0
scr20	110030	**110030**	0.0	117482	0.0	**110030**	0.0
sko42	15812*	**15812**	0.9	16090	1.7	**15812**	0.1
sko49	23386*	**23386**	1.3	24018	4.1	23394	0.5
sko56	34458*	**34458**	82.1	35540	4.9	34478	0.5
sko64	48498	**48498**	185.9	49546	13.4	48512	1.0
sko72	66256*	**66256**	411.1	67876	26.6	66306	2.5
sko81	90998*	**90998**	2696.2	93838	20.4	91038	17.1
sko90	115534*	**115534**	1262.7	118970	44.1	115610	2.5
sko100a	152002*	**152026**	533.8	156144	55.3	152114	10.0
sko100b	153890*	**153890**	3157.4	158124	78.5	154060	27.7
sko100c	147862*	**147862**	1240.9	151412	75.9	147890	20.1
sko100d	149576*	**149576**	982.6	154316	25.8	149746	23.2
sko100e	149150*	**149150**	3018.3	153766	72.5	149270	18.8
sko100f	149036*	**149036**	6259.9	154190	60.2	149218	20.5
ste36a	9526	**9526**	0.4	9888	1.5	**9526**	0.1
ste36b	15852	**15852**	0.1	16528	1.1	**15852**	0.3
ste36c	8239110	**8239110**	25.1	8425518	1.0	**8239110**	0.3
tai10a	*	**135028**	0.0	135028	0.0	**135028**	0.0
tai10b	*	**1183760**	0.0	1183760	0.0	**1183760**	0.0
tai12a	224416	**224416**	0.0	229092	0.0	**224416**	0.0
tai12b	39464925	**39464925**	0.0	40376360	0.0	**39464925**	0.0
tai15a	388214	**388214**	0.0	391744	0.0	**388214**	0.0
tai15b	51765268	**51765268**	0.0	51983343	0.0	**51765268**	0.0
tai17a	491812	**491812**	0.0	508078	0.0	**491812**	0.0
tai20a	703482	**703482**	0.0	725536	0.0	**703482**	0.0
tai20b	122455319	**122455319**	0.0	123727842	0.0	**122455319**	0.0
tai25a	*	**1167256**	0.1	1206174	0.2	**1167256**	1.1
tai25b	344355646	**344355646**	0.0	345221478	0.2	**344355646**	0.1
tai30a	*	**1818146**	1.0	1884472	0.7	**1818146**	0.5
tai30b	637117113	**637117113**	2.5	6418335531	0.3	**637117113**	0.1
tai35a	*	**2422002**	18.1	2539314	0.6	2426164	0.0
tai35b	283315445	**283315445**	4.1	286717437	1.5	**283315445**	0.5
tai40a	*	**3141702**	194.2	3331068	1.9	3154696	3.3
tai40b	637250948	**637250948**	1.0	638061679	1.6	**637250948**	1.0
tai50a	*	**4956824**	807.1	5392710	5.7	4985686	0.4
tai50b	458821517	**458821517**	36.6	464193360	2.4	**458821517**	6.6
tai60a	7208572	**7238844**	100.0	7929858	5.8	7272336	9.0
tai60b	608215054	**608215054**	281.7	611817881	12.2	**608590490**	1.5
tai64c	1855928	**1855928**	0.3	1863794	2.7	**1855928**	2.0

Problem Instance	Optimal	BCO		Fast Ant		Ro-TS	
		Result	t_{best}	Result	t_{best}	Result	t_{best}
tai80a	13557864	13574000	3837.3	14773678	11.3	13656426	17.3
tai80b	818415043	818415043	2545.1	853009972	25.1	820396926	15.5
tai100a	21125314*	21145560	2348.6	23013322	18.6	21273006	17.9
tai100b	1185996137*	1186800864	10546.5	1238862965	57.6	1190084262	30.2
tho30	149936	**149936**	0.0	153244	0.4	**149936**	0.0
tho40	*	**240516**	2.9	251064	1.9	**240516**	1.4
tho150	8133398*	8137158	7906.3	8413890	226.1	8144742	29.9
wil50	*	**48816**	53.3	49070	3.4	48824	3.4
wil100	273038*	273046	8786.8	277702	70.6	273316	4.3

Table 4.1.: Results of the 3 algorithms on QAPLIB problems.

One thing that is immediately obvious in Table 4.1 is the runtime for Bee Colony Optimization is higher than for either of the other two algorithms. Table 4.2 shows the results found by BCO when only allowed the runtime used by the other algorithms.

Problem Instance	RoTS	BCO	Fast Ant	BCO
	Best solution	Best in RoTS time	Best Solution	Best in Fast ant time
sko100a	152114	152820	156144	152310
sko100b	154060	154752	158124	154676
sko100c	147890	148842	151412	148368
sko100d	149746	150392	154316	150334
sko100e	149270	150238	153766	149708
sko100f	149218	149716	154190	149702
tai100a	21273006	21295918	23013322	21295918
tai100b	1190084262	1196144831	1238862965	1196144831
tho150	8144742	8176752	8413890	8157830
wil100	273316	273958	277702	273596

Table 4.2.: A comparison of results attained by Bee Colony Optimization when allowed the runtime used by the best runs of Fast Ant System and Robust Tabu-Search.

4.5. Conclusions and Further Work

We have presented a Bee Colony Algorithm for the Quadratic Assignment Problem and compared it with well known and effective algorithms for the problem, running on almost all problems in the QAPLIB. The results showed the algorithm was capable of finding the optimal or best known solution on almost all of the problems we tested on and produced better solutions than Fast Ant System or Robust Tabu Search on most problems. One obvious downside with the algorithm is the runtime used on local search as problem size increases, simple profiling on runs shows the share of the runtime decreases from 99% local search and 1% construction on small problems, to 95% local search on larger problems (this is for the problem instance esc128, though on average over 8 problems of size 90 or above the local search usage was still 98% according to Java VisualVM sampler). However this does not account for the difference in runtime growth between BCO and RoTS, since the majority of runtime is still the RoTS runs. The main reason therefore is likely the number of RoTS iterations used in the algorithms. The RoTS algorithm was set to run a fixed 100000 iterations on each problem. By comparison, the BCO algorithm runs 300 * number of bees (set to 50 in the above results), which is already 15000 iterations of RoTS per iteration of bee construction. On smaller problems, BCO finds the optimal solution often in the first iteration of bee construction, so the runtime is very fast. For larger problems, it can find good results in a couple of iterations (within 1% of the optimal solution), but the best result isn't found until the 200th iteration of bee construction (or later), which means it has already run 3-4.5 million iterations of RoTS in total (an example is for tho150 it took 667 iterations to find the best result, which means 10005000 iterations of RoTS).

Table 4.2 shows direct comparisons between RoTS and BCO, and Fast Ant and BCO after the same runtime. BCO seems to be a much stronger algorithm than

Fast Ant, however the difference in runtime between RoTS and BCO to get similar results shows that despite its ability to find good solutions, these solutions perhaps take too long to find to be useful.

As a first attempt to reduce runtime of the algorithm, we explored changing the colony size in the hope we can get the same results in similar numbers of iterations. Reducing the colony size from 50 to 20 means we reduce the number of iterations of RoTS by 60%. However, results on some large problems suggested that the result quality worsened and the runtime did not improve significantly with these changes. Due to this apparently loss of solution quality with this smaller colony size, we increased the number of iterations of local search per solution from 300 to 1000, and reduced the maximum number of BCO iterations that would occur. The result of this change was better results earlier in the runtime of the algorithm (so the results found in the same time as RoTS runs were closer) but overall little change from the best results found using 50 bees, or t_{best}.

Table 4.3 highlights exactly what proportion of runtime is used by the RoTS local search during a BCO run. It shows that the proportion used by RoTS local search is high on all instances, though it appears the percentage decreases slightly as the size of the instance increases.

Problem Instance	% Local search	% Construction	% Other
sko100a	98.2	1.7	0.1
sko100b	98.5	1.4	0.1
tho150	97.7	2.2	0.1
wil50	99.1	0.7	0.2

Table 4.3.: Percentage runtime used in various sections of the BCO algorithm according to Java VisualVM's sampler

I tested a very simple idea to attempt to reduce this by only allowing bees that find a solution within a factor of x of the best solution found so far run local search

after solution construction, using x values of 1.1 and 1.2. However, neither of these significantly affected the runtimes, so further work could focus on finding a method that can reduce the runs of local search on bad quality solutions, similar to the Frequency Based Pruning Strategy for the TSP, and consequently improve runtimes for BCO on the QAP.

A. A selection of graphs used in the FireFighting experiments

A.1. Introduction

These graphs are represented by adjacency lists, and the root node is node number 0 in each case.

27 nodes 35 edges

0:1,4,17

1:0,2,14,20,23,24

2:1,5,13,16

3:11

4:0,8

5:2,15

6:16,20

7:9,20

8:4,9,21,22

9:7,8,17,20

10:21

11:3,15,17,22

12:23

13:2

14:1,15,25

15:5,11,14,16

16:2,6,15

17:0,9,11,19,22,26

18:26

19:17

20:1,6,7,9

21:8,10

22:8,11,17

23:1,12

24:1

25:14

26:17,18

33 nodes 59 edges

0:3,23
1:11,16,20
2:11,17,22,25
3:0,18,29,30,32
4:5,23
5:4,8,16
6:16,17,20
7:9,13,23,30
8:5,19,20
9:7,16,31
10:16,23,25
11:1,2,15,18,21
12:18,21
13:7,17,23,31
14:16,26
15:11,24
16:1,5,6,9,10,14,20,28,32
17:2,6,13,25,32
18:3,11,12,22,23
19:8,23,29
20:1,6,8,16,21,29
21:11,12,20
22:2,18
23:0,4,7,10,13,18,19
24:15,28,29
25:2,10,17,26
26:14,25
27:29,32
28:16,24
29:3,19,20,24,27
30:3,7
31:9,13,32
32:3,16,17,27,31

55 nodes 104 edges

0:8,19,42,46
1:5,27
2:13,45
3:7,39,47
4:11,27,51
5:1
6:23,37,48
7:3,24,25,29,33,43
8:0,9,12,19
9:8,15,36,40,45

10:14,18,32

11:4,32

12:8,15,24,30,35,43,51,53

13:2,21,48

14:10

15:9,12,31,34,46

16:19,38,40,42,49,51

17:33,51

18:10

19:0,8,16

20:24,38,51,54

21:13,25,26,27,37,54

22:39,41,52

23:6,39

24:7,12,20,31,45,46

25:7,21,27,30,44,49

26:21,28,30,37,38

27:1,4,21,25,32,45,50

28:26,37,41

29:7,36,43,51

30:12,25,26,41

31:15,24,34,52

32:10,11,27,35,37,43

33:7,17,38

34:15,31,40,49

35:12,32,40,43

36:9,29

37:6,21,26,28,32,50

38:16,20,26,33

39:3,22,23,48

40:9,16,34,35,52

41:22,28,30,46,52

42:0,16

43:7,12,29,32,35

44:25,47

45:2,9,24,27

46:0,15,24,41

47:3,44

48:6,13,39,53

49:16,25,34,51

50:27,37

51:4,12,16,17,20,29,49

52:22,31,40,41

53:12,48

54:20,21

63 nodes 106 edges

0:9,45,56

1:60

2:11,14,20,40

3:46,56

4:5,38,47,55

5:4,12,23,35

6:9,44

7:11,28

8:10,26,59

9:0,6,16,53

10:8,20,43,46,49,52

11:2,7,30,45,58

12:5,32,50,60

13:20,26,36,38

14:2,15

15:14

16:9,56,59

17:20

18:19,20,35

19:18,39

20:2,10,13,17,18,34,51,58

21:23,39,60

22:33,59

23:5,21

24:25,28,56

25:24,26,31,44,45,52,58

26:8,13,25

27:35

28:7,24,44,60

29:49,53

30:11,34,51,60

31:25

32:12,45

33:22,38,57,59

34:20,30

35:5,18,27,36,49,58

36:13,35,42,53

37:39,59

38:4,13,33,47,48

39:19,21,37

40:2,42

41:50,58

42:36,40,54,61

43:10,51,57,60

44:6,25,28

45:0,11,25,32,48,61

46:3,10,48,54

47:4,38

48:38,45,46,56
49:10,29,35,56
50:12,41,60
51:20,30,43
52:10,25
53:9,29,36,62
54:42,46,58,59
55:4
56:0,3,16,24,48,49
57:33,43
58:11,20,25,35,41,54
59:8,16,22,33,37,54
60:1,12,21,28,30,43,50
61:42,45,62
62:53,61

102 nodes 175 edges

0:11,36,38,67
1:80,90
2:41,74
3:70,76
4:48,52,60,66,89,97
5:38,66,81
6:35,53,69
7:38
8:38,82,85,98
9:30,44,48,54,71
10:47,49,54,68
11:0,46,70
12:16,52,59,63,64,67,101
13:34,77
14:47,60
15:29,51,79,83,87
16:12
17:23,30,35,77
18:38,83,88,92
19:37,43,61
20:32,63,68
21:24,54,55
22:49,76,81
23:17,47
24:21,34,46,81,91
25:69,94
26:29,37,40,61,95
27:28,55,60,70
28:27
29:15,26,48,76

30:9,17
31:41,72
32:20,43,56
33:53,62,70,85,87
34:13,24,56,81
35:6,17,57,59,90
36:0,47
37:19,26,41,83
38:0,5,7,8,18,75,82
39:56
40:26,45,58,64,101
41:2,31,37,51,55
42:88,91,101
43:19,32,46,68,69,76,94,96
44:9,49,100
45:40,60,64,86,93
46:11,24,43,67
47:10,14,23,36
48:4,9,29,61,88
49:10,22,44,77
50:54,61,63,73,98
51:15,41
52:4,12,64
53:6,33,73
54:9,10,21,50,60,79,90
55:21,27,41,57,92
56:32,34,39,78,86
57:35,55
58:40,64
59:12,35,98
60:4,14,27,45,54,74,100
61:19,26,48,50,78
62:33,84,92
63:12,20,50
64:12,40,45,52,58,84,88
65:76
66:4,5,67,99
67:0,12,46,66,93,95,101
68:10,20,43
69:6,25,43
70:3,11,27,33,74
71:9,74
72:31,88
73:50,53,98
74:2,60,70,71
75:38
76:3,22,29,43,65
77:13,17,49,84,101
78:56,61
79:15,54

80:1
81:5,22,24,34,90
82:8,38
83:15,18,37
84:62,64,77
85:8,33
86:45,56
87:15,33
88:18,42,48,64,72,99
89:4
90:1,35,54,81
91:24,42
92:18,55,62
93:45,67
94:25,43
95:26,67
96:43,97
97:4,96
98:8,50,59,73
99:66,88
100:44,60
101:12,40,42,67,77

113 nodes 199 edges

0:24,65
1:22
2:3,4,38,46,56,86
3:2
4:2,11,12,72
5:10,22,47,56,60,111
6:16,20,45,74,99,105
7:33,36,54,82,107
8:71,88
9:25,33,71
10:5,14,25,104
11:4,40,57,64
12:4,17,19
13:30,36,49,63,75
14:10,24,73,80
15:76,95,99
16:6,50,51,60,68,70,89
17:12,41,43,44,46,50,65,81,86,92,97,99,100,105
18:21,80,102
19:12,26,28,31,46,82,105
20:6,62
21:18
22:1,5,26,51,71,97

23:76
24:0,14,39,101
25:9,10,50,59,62,91
26:19,22,78
27:29,86
28:19,83
29:27,55
30:13,72
31:19,74,77,105
32:60
33:7,9
34:35,57,97
35:34,39,43,88,89,94
36:7,13,74
37:96
38:2,54
39:24,35,46,88
40:11,46,77,90
41:17
42:88
43:17,35,70
44:17
45:6,49
46:2,17,19,39,40,66,90,96
47:5,48,101
48:47,78
49:13,45
50:16,17,25,68,78,106
51:16,22
52:61,75,104
53:94,106,109
54:7,38,59,84
55:29,61,73,112
56:2,5,77
57:11,34
58:100,101,112
59:25,54,67,69,88
60:5,16,32
61:52,55,80,85,93,106,110
62:20,25,78,80,100,107
63:13,100
64:11,107,112
65:0,17,87,109
66:46,76,95,107
67:59,77,94
68:16,50,87,91
69:59,100
70:16,43,107
71:8,9,22,87
72:4,30,75

73:14,55,84
74:6,31,36
75:13,52,72
76:15,23,66,79,92
77:31,40,56,67
78:26,48,50,62
79:76,105
80:14,18,61,62,98
81:17,99
82:7,19,101
83:28,112
84:54,73,103,106
85:61
86:2,17,27,105,111
87:65,68,71,103
88:8,35,39,42,59,104,110
89:16,35
90:40,46
91:25,68,94,101
92:17,76
93:61,108
94:35,53,67,91,112
95:15,66
96:37,46,102,104
97:17,22,34,101
98:80
99:6,15,17,81
100:17,58,62,63,69
101:24,47,58,82,91,97,102,108
102:18,96,101
103:84,87
104:10,52,88,96
105:6,17,19,31,79,86
106:50,53,61,84,112
107:7,62,64,66,70
108:93,101
109:53,65
110:61,88
111:5,86
112:55,58,64,83,94,106

139 nodes 272 edges

0:58,99
1:10,62,109,110
2:47,64,102,119
3:33,117
4:48,100,135

5:78,138
6:45,81,119
7:8,40
8:7,56,79,111,136
9:25,40,120
10:1,91,106,109
11:28,67,88,112,117
12:41,60,74
13:54,128
14:34,79,81
15:36,49,73,91,97,117,124
16:27
17:136,138
18:22,27,37,52,119
19:41,44,62,95,109
20:124,127,132
21:59,83,89,104,106,113,123
22:18,29,61,64,76,80,96,102,109,114,127,136,137
23:41,113,128
24:56,66
25:9,42,44
26:41,68,78,90
27:16,18,59,87,112,127
28:11,32,107,120
29:22,80,128
30:59,126
31:71,89,105
32:28,114
33:3,38,58,109
34:14,38,50,105,109
35:125,131
36:15,37,61,92,93
37:18,36,54,92,108,109
38:33,34,93,128
39:40,111,117
40:7,9,39,43,63,128
41:12,19,23,26
42:25,59,67,125,127
43:40,46,63,78
44:19,25,61,99
45:6,108,137
46:43,78
47:2,138
48:4,59
49:15,50,136
50:34,49,79,85,126
51:66,76,82,135
52:18
53:61,77,85
54:13,37,64,65,134

55:64,67,80,84
56:8,24,80,86,119,130
57:61,102,126
58:0,33,83,118,130
59:21,27,30,42,48,93
60:12,84,104,138
61:22,36,44,53,57,72,98,103
62:1,19
63:40,43,65
64:2,22,54,55,65
65:54,63,64,80,83,85,87,97
66:24,51,81,89,116,130
67:11,42,55,73,88
68:26
69:131
70:74,82,101,112
71:31,100,115,134
72:61,109
73:15,67,114
74:12,70,121
75:100
76:22,51,100
77:53,86,97
78:5,26,43,46,100
79:8,14,50,120
80:22,29,55,56,65,117,118
81:6,14,66,90,125
82:51,70,123
83:21,58,65,84,124
84:55,60,83
85:50,53,65,99,105
86:56,77,124,133
87:27,65
88:11,67,113,125,126
89:21,31,66,133
90:26,81,126,130
91:10,15,97,102,130,138
92:36,37,114
93:36,38,59,107
94:131
95:19
96:22
97:15,65,77,91
98:61,129
99:0,44,85,132
100:4,71,75,76,78,116,136
101:70,106,112
102:2,22,57,91,110
103:61
104:21,60

105:31,34,85,120
106:10,21,101,112
107:28,93,123,135
108:37,45,118
109:1,10,19,22,33,34,37,72,128,131
110:1,102
111:8,39,127
112:11,27,70,101,106,132,136,138
113:21,23,88,129
114:22,32,73,92,136
115:71,120
116:66,100,121,126,133,134
117:3,11,15,39,80,128
118:58,80,108
119:2,6,18,56
120:9,28,79,105,115,137
121:74,116
122:124
123:21,82,107
124:15,20,83,86,122
125:35,42,81,88,129
126:30,50,57,88,90,116
127:20,22,27,42,111
128:13,23,29,38,40,109,117
129:98,113,125
130:56,58,66,90,91
131:35,69,94,109
132:20,99,112
133:86,89,116
134:54,71,116
135:4,51,107,137
136:8,17,22,49,100,112,114
137:22,45,120,135
138:5,17,47,60,91,112

157 nodes 217 edges

0:118,128,138
1:31,37
2:36,72
3:120,140,141
4:49,61,63
5:28,53,69,73,84,97,124,133,141,142,151
6:23,28,56
7:24,148
8:87
9:21,74,77
10:106,133,153

11:36,82,120,122
12:134
13:43,60,61
14:22,151
15:21,140
16:40,64
17:26,106,136
18:36
19:124
20:105,153
21:9,15,78,153
22:14,98,136,144
23:6,53
24:7,30
25:100,128
26:17,57,104,149
27:63
28:5,6,45,109,123
29:69,154
30:24,71,112,138,144
31:1,106,138
32:112
33:48,121,123
34:38
35:125
36:2,11,18,53,94,149
37:1,44,57,72
38:34,94
39:115
40:16,71
41:102
42:115,154
43:13,47,62,115
44:37,45,109
45:28,44,118
46:85
47:43,134,145
48:33,64,66
49:4,80
50:142,153
51:69,103
52:65,145
53:5,23,36,68,90
54:78,127,130,150
55:121
56:6
57:26,37,59,91,136,141
58:64,143,147
59:57,101
60:13

61:4,13
62:43
63:4,27,89,155
64:16,48,58,87,100,118,134,151
65:52,101
66:48
67:105
68:53,137,146
69:5,29,51,80,131
70:83,154
71:30,40,108,111,117
72:2,37,112
73:5,76,89
74:9,95
75:117,133
76:73,80
77:9,140
78:21,54
79:105,144,156
80:49,69,76
81:148
82:11,103,149
83:70,109
84:5,88,129
85:46,86,94,152
86:85,132
87:8,64,98,156
88:84
89:63,73
90:53,107,111,118
91:57,94
92:109
93:99,153
94:36,38,85,91,136,151,155
95:74,98,101
96:119
97:5
98:22,87,95,149
99:93
100:25,64
101:59,65,95,131,141
102:41,136
103:51,82,110
104:26
105:20,67,79
106:10,17,31
107:90
108:71
109:28,44,83,92,131,141
110:103,126,143

111:71,90,134,143
112:30,32,72,134,136
113:151
114:147
115:39,42,43,139,144
116:134
117:71,75
118:0,45,64,90
119:96,136
120:3,11,156
121:33,55
122:11
123:28,33
124:5,19
125:35,156
126:110,136,148,152
127:54
128:0,25,130
129:84
130:54,128
131:69,101,109
132:86
133:5,10,75,154
134:12,47,64,111,112,116,139
135:146
136:17,22,57,94,102,112,119,126
137:68
138:0,30,31
139:115,134,146
140:3,15,77
141:3,5,57,101,109
142:5,50
143:58,110,111,148
144:22,30,79,115
145:47,52,154
146:68,135,139
147:58,114
148:7,81,126,143
149:26,36,82,98
150:54
151:5,14,64,94,113
152:85,126
153:10,20,21,50,93
154:29,42,70,133,145
155:63,94
156:79,87,120,125

Bibliography

[BR84] R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2): 169–174, August 1984, http://ideas.repec.org/a/eee/ejores/v17y1984i2p169-174.html.

[Bra01] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2): 163–177, 2001.

[CFvL12] Marek Cygan, Fedor V Fomin, and Erik Jan van Leeuwen. Parameterized complexity of firefighting revisited. In *Parameterized and Exact Computation*, pages 13–26. Springer, 2012.

[DS04] Marco Dorigo and Thomas Stutzle. *Ant Colony Optimization*. MIT Press, 2004.

[FJF93] Charles Fleurent, Jacques, and Jacques A. Ferland. Genetic hybrids for the quadratic assignment problem. In *DIMACS Series in Mathematics and Theoretical Computer Science*, pages 173–187. American Mathematical Society, 1993.

[FM09] Stephen Finbow and Gary MacGillivray. The firefighter problem: a survey of results, directions and questions. *The Australasian Journal of Combinatorics*, 43: 57, 2009.

[Gil62] P. C. Gilmore. Optimal and suboptimal algorithms for the quadratic assignment problem. *Journal of the Society for Industrial and Applied Mathematics*, 10(2): pp. 305–313, 1962, http://www.jstor.org/stable/2099107.

[KM10] Andrew King and Gary MacGillivray. The firefighter problem for cubic graphs. *Discrete Mathematics*, 310: 614–621, 2010.

[LT01] Panta Lucic and Dusan Teodorovic. Bee system: modeling combinatorial optimization transportation engineering problems by swarm intelligence. In *Preprints of the TRISTAN IV Triennial Symposium on Transportation Analysis*, pages 441–445, 2001.

[MC99] V. Maniezzo and A. Colorni. The ant system applied to the quadratic assignment problem. *Knowledge and Data Engineering, IEEE Transactions on*, 11(5): 769 –778, sep/oct 1999.

[MDBS98] Vittorio Maniezzo, Vittorio Maniezzo Scienze Dell'informazione, University Di Bologna, and Via Sacchi. Exact and approximate nondeterministic tree-search procedures for the quadratic assignment problem, 1998.

[MW03] Gary MacGillivray and P. Wang. On the firefighter problem. *Journal of Combinatorial Mathematics and Combinatorial Computing*, 47: 83–96, 2003.

[New08] MEJ Newman. The mathematics of networks. *The New Palgrave Encyclopedia of Economics*, -: –, 2008.

[PBMW99] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[QAP] Qaplib website.

[Stü97] Thomas Stützle. Max-min ant system for quadratic assignment problems, 1997.

[Tai91] Éric D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(4-5): 443–455, 1991, http://mistic.heig-vd.ch/taillard/articles.dir/Taillard1991.pdf.

[Tai95] Éric D. Taillard. Comparison of iterative searches for the quadratic assignment problem. *Location Science*, 3(2): 87 – 105, 1995, http://mistic.heig-vd.ch/taillard/articles.dir/Taillard1995.pdf. Old technical report CRT989.

[Tai98] Éric D. Taillard. Fant: Fast ant system. Technical report, Istituto Dalle Molle Di Studi Sull Intelligenza Artificiale, 1998. IDSIA Technical Report IDSIA-46-98.

[TDS11] D. Teodorovic, T. Davidovic, and M. Selmic. Bee colony optimization: the applications survey. *ACM Transaztions on Computational Logic*, 1529: 3785, 2011.

[vF74] Karl von Frisch. Decoding the language of the bee. *Science*, 185(4152): pp. 663–668, 1974, http://www.jstor.org/stable/1738718.

[WLC08a] Li-Pei Wong, M.Y.H. Low, and Chin Soon Chong. A bee colony optimization algorithm for traveling salesman problem. In *Modeling Simulation, 2008. AICMS 08. Second Asia International Conference on*, pages 818 –823, may 2008.

[WLC08b] Li-Pei Wong, M.Y.H. Low, and Chin Soon Chong. Bee colony optimization with local search for traveling salesman problem. In *Industrial Informatics, 2008. INDIN 2008. 6th IEEE International Conference on*, pages 1019 –1025, july 2008.

[WLC09a] Li-Pei Wong, M.Y.H. Low, and Chin Soon Chong. A bee colony optimization algorithm with the fragmentation state transition rule for the traveling salesman problem. In *Proceedings of the 4th Virtual International Conference on Intelligent Production Machines and Systems (IPROMS)*, 2009.

[WLC09b] Li-Pei Wong, M.Y.H. Low, and Chin Soon Chong. An efficient bee colony optimization algorithm for traveling salesman problem using frequency-based pruning. In *Industrial Informatics, 2009. INDIN 2009. 7th IEEE International Conference on*, pages 775 –782, june 2009.